

RGIssearch: A C++ program for the determination of Renormalization Group Invariants

Rob Verheyen^{1,*}

Abstract

RGIssearch is a C++ program that searches for invariants of a user-defined set of renormalization group equations. Based on the general shape of the β -functions of quantum field theories, RGIssearch searches for several types of invariants that require different methods. Additionally, it supports the computation of invariants up to two-loop level. A manual for the program is given, including the settings and set-up of the program, as well as a test case.

Keywords: Renormalization, Computer Algebra, Sparse Linear Systems

*Corresponding author

Email address: RobVerheyen@gmail.com (Rob Verheyen)

¹IMAPP, Mailbox 79, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Program Summary *Manuscript Title:* RGIsearch: A C++ program for the determination of Renormalization Group Invariants

Authors: Rob Verheyen

Program Title: RGIsearch

Journal Reference:

Catalogue identifier:

Licensing provisions: none

Programming language: C++

Computer: Desktop PC

Operating system: Linux

RAM: Ranging from several MB to several GB

Keywords: Renormalization, Computer Algebra, Sparse Linear Systems

Classification: 2.9 Theoretical Methods, 4.8 Linear Equations and Matrices, 5 Computer Algebra

Nature of problem: The determination of renormalization group invariants, which can be used as a probe for the high-energy behaviour of theories in particle physics.

Solution method: Several types of invariants are considered based on the general shape of the renormalization group equations. The problem of computing these invariants is then reduced to creating and solving very large systems of linear equations or generalized eigenvalue problems. Using the specific sparsity structure of these systems, the systems can be solved.

Restrictions: Since the algorithms that solve the linear systems and generalized eigenvalue problems are very efficient, the main restriction is the amount of available RAM, where the systems are stored. The program supports polynomial renormalization group equations, which are typical particle physics, up to two-loop order.

Running time: Ranging from seconds to minutes.

1. Introduction

Modern theories in particle physics, such as supersymmetry, often predict new physics at experimentally inaccessible energy scales. The high energy behavior of such a theory can be probed by considering the evolution of its parameters, which were measured at some experimental scale, to higher energies using the *renormalization group* (RG) equations. Low-energy parameters can be translated to high-energy ones with several methods, each with their own up- and downsides [1]. One of these methods uses RG *invariants*, combinations of the parameters of a theory that are invariant under the renormalization group flow [2, 3]. While this method offers multiple advantages over others, finding such invariants is highly nontrivial, especially at high loop orders or for theories with large parameter spaces. RGIsearch is a C++ program that is able to compute these invariants for any set of RG equations. In this article, the methods that RGIsearch uses to find invariants are described and a manual is provided. Obviously, checking if the proposed invariants produced by the program are actually invariant is trivial. RGIsearch was recently used to compute several new invariants of various versions of the Minimal Supersymmetric Standard Model [4].

2. Methodology

The RG equations are differential equations for the parameters of a theory as function of the energy scale μ . Consider the RG equation of a renormalized theory for a parameter $x(\mu)$. The corresponding β -function is defined as:

$$\beta(x) = 16\pi^2 \frac{dx}{dt}, \quad (1)$$

where $t = \log_{10}(\mu/\mu_0)$. Since the β -functions of quantum field theories are polynomials in the parameters of the theory, RGIsearch considers several types of polynomial RG invariants. We first discuss methods for these types for one-loop β -functions, where in quantum field theories the coefficients of these polynomials are rational numbers, before extending it to two loops.

2.1. Monomial Invariants

Monomial invariants assume the form:

$$M = \prod_{i=1}^n x_i^{a_i}, \quad (2)$$

where x_i are the n parameters of the theory and $\vec{a} \in \mathbb{Z}^n$. The powers \vec{a} can be taken to be integers since the coefficients of the β -functions are rational, and any power of a monomial invariant is still a monomial invariant. Taking the derivative and dividing out M , the condition for invariance is:

$$\sum_{i=1}^n \frac{a_i \beta(x_i)}{x_i} = 0. \quad (3)$$

The left-hand side of eq. (3) is a polynomial which is required to vanish for all values of all x_i . This is only possible if all monomial terms are cancelled internally. Therefore, eq. (3) can be reduced to a relatively small linear system of equations on the powers a_i , where the equations represent the requirement of the cancellation of all monomial terms. Every element in the nullspace of this linear system represents a monomial invariant. By finding the basis vectors for the nullspace, the independent monomial invariants are determined. The other elements of the nullspace are then just products of powers of these invariants.

2.2. Dimensionalities

Before extending the above method to polynomial invariants, the concept of *dimensionalities* has to be introduced. A dimensionality D is represented by a vector $\vec{D} \in \mathbb{Z}^n$ such that:

$$\dim_D(x_i) = D_i. \quad (4)$$

Dimensionalities are additive for monomials:

$$\dim_D \left(\prod_{i=1}^n x_i^{a_i} \right) = \vec{a} \cdot \vec{D}, \quad (5)$$

where \cdot represents the standard Euclidean inner product. A system of β -functions is said to have dimensionality D if:

$$\forall i \in [1, \dots, n] : \dim_D(\beta(x_i)) - \dim_D(x_i) = c_D, \quad (6)$$

where the dimensionality of a polynomial such as the β -functions means that all included monomial terms have the same dimensionality, and $c_D \in \mathbb{Z}$ is a constant. A system of β -functions can have multiple dimensionalities. Note that these dimensionalities are similar to the well-known physical dimensionalities, such as mass dimension. In fact, mass dimension is typically

one of the dimensionalities of a system of β -functions with $c_{D_m} = 0$. RGIssearch finds the dimensionalities of a system of β -functions by reducing the problem to a different system of linear equations. Writing out the β -functions in their monomial terms:

$$\beta(x_i) = \sum_{j=1}^{m_i} C_{ij} M_{ij} \text{ where } M_{ij} = \prod_{k=1}^n x_k^{b_{ij,k}}, \quad (7)$$

where the m_i are the number of monomial terms in $\beta(x_i)$. Eq. (6) can now be written as:

$$\forall i \in [1, \dots, n], j \in [1, \dots, m_i]: \vec{b}_{ij} \cdot \vec{d} - D_i - c_D = 0, \quad (8)$$

which is a linear system in \vec{d} and c_D and can thus easily be solved.

2.3. Polynomial Invariants

Similar to the β -functions, polynomial invariants assume the form:

$$P = \sum_{i=1}^m C_i M_i \text{ where } M_i = \prod_{j=1}^n x_j^{a_{i,j}}, \quad (9)$$

where $\vec{C} \in Z^m$ and $\vec{a}_i \in Z^n$. Eq. (6) implies:

$$\dim_D \left(\frac{d}{dt} M_i \right) = \dim_D (M_i) + c_D. \quad (10)$$

This means that there cannot be any overlap between derivatives of monomial terms with different dimensionalities. Therefore, it suffices to consider polynomial invariants with monomial terms of the same dimensionalities. Thus, RGIssearch first computes the dimensionalities of the system. Next, if r dimensionalities are found, it runs through a user-defined range of dimensionalities $\vec{d} \in \mathbb{Z}^l$, computing the set:

$$\mathcal{M}_p(\vec{d}) = \left\{ \prod_{i=1}^n x_i^{a_i} \mid \forall l \in [1, \dots, r]: \dim_l \left(\prod_{i=1}^n x_i^{a_i} \right) = d_l, -p \leq x_i \leq p \right\}, \quad (11)$$

where p is a user-defined setting controlling the size of $\mathcal{M}_p(\vec{d})$. The polynomial invariant then becomes:

$$P(\vec{d}) = \sum_{i=1}^{|\mathcal{M}_p(\vec{d})|} C_i M_i \text{ where } M_i \in \mathcal{M}_p(\vec{d}). \quad (12)$$

The requirement for invariance now is:

$$\sum_{i=1}^m C_i \frac{d}{dt} M_i = 0, \quad (13)$$

which can be converted to a linear system of equations in \vec{C} by demanding that eq. (13) holds for all values of all parameters. While the systems encountered previously were relatively small, the size of this system heavily depends on the size of \mathcal{M}_p , which should be taken as large as possible to reach more invariants. The associated system of equations can grow very large, but it can still be solved quickly due to its sparsity.

The elements of the nullspace make up all possible invariants that can be constructed from the set \mathcal{M}_p . By computing the basis vectors of the nullspace, RGIsearch finds all independent invariants. The rest of the nullspace consists of linear combinations of these invariants. In addition, products of these invariants can be found at other dimensionalities. After running through the range of user-defined dimensionalities, a simple filtering algorithm searches for these products of invariants and removes them from the output.

2.4. Factorization

RGIsearch supports a minor extension to the method of the previous section, where a single variable can be factorized from a polynomial invariant:

$$P_j(\vec{d}) = x_j^b \left(\sum_{i=1}^m C_i M_i \right) \text{ where } M_i \in \mathcal{M}_p(\vec{d}). \quad (14)$$

Taking the derivative and dividing out x_j^b , the requirement for invariance is:

$$\frac{b\beta(x_j)}{x_j} \sum_{i=1}^m C_i M_i + \sum_{i=1}^m C_i \frac{d}{dt} M_i = 0. \quad (15)$$

Instead of a linear system, eq. (15) leads to a system of the form:

$$\mathbf{A} + b\mathbf{Q} = 0. \quad (16)$$

This can be interpreted as a nonsquare, generalized eigenvalue problem of a similar size as regular polynomial searching. It is approached as a regular eigenvalue system by solving the eigenvalues b first, after which the system becomes linear.

2.5. System Creation

The methods described in subsections 2.3 and 2.4 require the creation of very large (typically $\mathcal{O}(10^5) \times \mathcal{O}(10^4)$ to $\mathcal{O}(10^6) \times \mathcal{O}(10^5)$) matrices. These matrices are constructed by grouping together monomial terms that appear in eq. (13) and eq. (15). RGIsearch performs this task by calculating these derivative monomial terms one-by-one and directly placing them into the matrix. To be able to do that, a `std::map` is used to make the translation from a monomial term to a row in the matrix.

Monomial terms are stored in a `std::vector` of small integer numbers indicating the parameters and their powers. Because multiple numbers are required to fully specify a monomial term, and because containers such as `vector` require some memory overhead², the *Cantor pairing function*:

$$P : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \text{ where } P(n_1, n_2) = \frac{1}{2}(n_1 + n_2)(n_1 + n_2 + 1) + n_2, \quad (17)$$

is used to convert monomial terms into a single natural number, which is then mapped to a row in the matrix. To pair more than two numbers, the `vector` is extended with zeroes until its size is a power of two. The pairing function is then applied pairwise until a single number remains. The Cantor pairing function is injective, so in principle there is no danger of collision. However, the resulting number might overflow its designated 128 bit integer. Even if this happens, the probability of a collision is negligibly small.³

²Containers without overhead exist, but their basic functions such as `size()`, which is heavily used throughout the algorithm, have longer computation times, slowing down the algorithm significantly.

³The calculation of this probability is comparable to the birthday problem. The probability of collision for a 128 bit integer and a matrix of typical size can be approximated to $1 - \exp(-10^{12}/2^{129})$

2.6. Markowitz Pivoting and Sparse Matrix Handling

Next, RGIsearch solves the systems that were created in the above procedure. To find the nullspace of these systems, RGIsearch uses fraction free Gaussian elimination with Markowitz pivoting. An implicit pivoting step is performed before every elimination step to preserve sparsity. This pivoting is based on the heuristic *Markowitz count* [5]:

$$(r_i - 1)(c_j - 1), \quad (18)$$

where r_i is the number of nonzero elements in row i and c_j the number of nonzero elements in column j . Before every elimination step, the nonzero element with the lowest Markowitz count is located and (implicitly) pivoted to the top-left. By performing this pivoting step, sparsity is preserved and the number of required operations is reduced as much as possible.

Note that the Markowitz count is zero for rows or columns with only a single nonzero element (singlets). These rows and columns are selected first, since they are very cheap to eliminate and produce no additional fill-in. The matrices that appear while computing polynomial invariants usually contain many row singlets, so the algorithm is specifically tailored to use these. It is abstractly shown below. The i th row of a matrix \mathbf{A} is denoted with A_i , and the matrix is viewed as a collection of these rows since the order of the rows can freely be changed in nullspace computations.

The reduction of a row simply means that all nonzero elements of the row are divided by their total greatest common divisor (gcd). If the algorithm terminates while \mathbf{A} is not yet empty, then all variables have been eliminated and no nullspace exists. If \mathbf{A} does empty, the nullspace can easily be solved by selecting a basis for the space of variables that have not been eliminated. The rest of the variables are then backwardly solved by the equations in \mathbf{B} .

For the method described in subsection 1.4, elimination of matrices of similar sizes containing univariate polynomials with integer coefficients has to be performed. The algorithm is very similar. The Markowitz count is extended to:

$$(r_i - 1)(c_j - 1)(d_{ij} - 1), \quad (19)$$

where d_{ij} is the degree of the matrix element at row i and column j . Furthermore, integer coefficients are maintained through fraction-free Gaussian elimination using the subresultant polynomial remainder sequence

Algorithm: Gaussian elimination with Markowitz pivoting

Input : A sparse matrix $\mathbf{A} \in \mathbb{Z}^{m \times n}$

Output: A fully eliminated matrix $\mathbf{B} \in \mathbb{Z}^{r \times n}$, a vector of flags that indicate elimination of variables $\vec{o} \in \mathbb{Z}_2^n$

$B \leftarrow \emptyset, Q \leftarrow \emptyset, \vec{o} \leftarrow (0, \dots, 0)$

for all $k \in [0, \dots, m]$ **do**

if A_k is a row singlet **then**

$Q \leftarrow Q \cup \{k\}$

while \mathbf{A} is not empty or \vec{o} still contains zeroes **do**

$i \leftarrow 0, j \leftarrow 0$

if $Q \neq \emptyset$ **then**

$i \leftarrow$ Some element of Q

$Q \leftarrow Q - \{i\}$

if A_i is a row singlet **then**

$j \leftarrow$ Column index of the nonzero element of A_i

else

 Find the nonzero element a_{mn} in \mathbf{A} minimizing

$(r_m - 1)(c_n - 1)$

$i \leftarrow m, j \leftarrow n$

if $j \neq 0$ **then**

$o_j \leftarrow 1$

$B \leftarrow B \cup \{A_i\}$

for all l with $a_{lj} \neq 0$ **do**

 Fraction-freely eliminate a_{lj} from A_l using row A_i

 Reduce A_p using the gcd of its elements

if A_l is empty **then**

$A \leftarrow A - \{A_l\}$

if A_l is a row singlet **then**

$Q \leftarrow Q \cup \{l\}$

$A \leftarrow A - \{A_i\}$

algorithm outlined in [6]. The eigenvalues are then computed by solving the polynomials that appear on the implicit diagonal of \mathbf{B} .

To improve speed and memory efficiency, RGIsearch uses the *list of lists* method to store matrices [5]. They are stored as a list of rows, containing all nonzero matrix elements which are tagged by their column index. While this method is not the most efficient of all sparse matrix storage techniques in terms of memory usage, it allows for very fast access to individual rows which is extremely important in the above algorithm. Additionally, insertion of new nonzero matrix elements is relatively cheap which is significant for the elimination steps. Fast access to columns is also important during the elimination steps. Therefore, a second matrix is stored as a list of columns instead of rows. This matrix is used purely as an indexing structure and thus only stores the row indices of nonzero elements. Both structures are updated simultaneously during the elimination procedure.

2.7. Two-loop invariants

RGIsearch allows for computation of regular polynomial invariants for two-loop β -functions. The general form of a two-loop β -function is:

$$\beta(x_i) = \beta^{(1)}(x_i) + \frac{1}{16\pi^2}\beta^{(2)}(x_i). \quad (20)$$

Therefore, a two-loop invariant looks like:

$$I = I_1 + \frac{1}{16\pi^2}I_2. \quad (21)$$

Taking the derivative and letting $I_{1,2}^{(j)}$ denote the part of the derivative involving $\beta^{(j)}(x_i)$, the condition for invariance for the one-loop and two-loop terms separate:

$$I_1^{(1)} = 0 \text{ and } I_1^{(2)} + I_2^{(1)} = 0. \quad (22)$$

The term $I_2^{(2)}$ is formally of three-loop order. These conditions are converted into a linear system in a very similar fashion to the one-loop case.

3. Usage

RGIsearch is available from a public repository [7]. The latest version of RGIsearch can be acquired with:

```
$ git clone https://github.com/rbvh/RGIsearch.git
```

3.1. Compilation

To compile, do:

```
$ make
```

This will generate all binary files and link them into an executable. This executable can then be run with:

```
$ ./RGIsearch
```

If any β -functions, settings or any of the sources are changed, `make` should update the binaries. If for some reason it does not, or it is otherwise required, one can do:

```
$ make clean  
$ make
```

3.2. Settings

The program settings can be found in the file `settings.h`.

```
const int DIM_SEARCH_PARAMETER
```

This constant controls the range of dimensionalities the algorithm searches. Depending on the dimensionalities that are found and the value of this constant, RGIsearch calculates a suitable selection of dimensionality vectors to search through.

```
const int MAX_TERM
```

This constant controls the number of different parameters allowed in a monomial term during the execution of the algorithms for polynomial invariants. For instance, if it is set to 2, terms like xy^2 can occur, but xyz is forbidden. Changing it can have huge influence of the required computation times and memory.

```
const int FILTER_THRESHOLD
```

This constant controls the filtering mechanism. If it is set to a higher value, the filtering algorithm will perform a more elaborate attempt to find all possible products of previously found invariants to compare against any newly found invariants. The default value should almost always be sufficient.

```
bool INCLUDE_TWO_LOOP
```

Set to true to search for invariants at two-loop level.

`bool FACTORIZE`

Set to true to include the algorithm that searches for factorized polynomial invariants. The code does currently not support factorization for two-loop level searches.

`bool REPORT`

Set to true to make the program report its activities in more detail.

3.3. β -Functions

The β -functions can be found in `equations.cpp`.

To define the β -functions of a theory, the parameters of the theory first need to be defined. This is done with:

```
Param newPar("parName", parSize);
```

`parSize` is the size of the parameter (1 for a scalar, n for a $n \times n$ matrix). If a parameter is complex, its daggered counterpart must be defined. This is done with:

```
Param newParDagger = dagger(newPar);
```

The `Param` class has several methods that simplify matrix parameters. They include:

1. `botRight()` sets all elements except for the bottom right component equal to zero.
2. `diag()` sets all offdiagonal elements equal to zero.
3. `botRightDiag()` sets all offdiagonal terms equal to zero, and makes all components except for the bottom right degenerate.

After defining the parameters, one can define the β -functions. These are stored as a vector of the objects `BetaFunc`:

```
vector<BetaFunc> nameOfBetaFuncs
```

The β -function of a parameter can then be defined as:

```
BetaFunc bNewPar(newPar);  
bNewPar = <polynomial>;  
nameOfBetaFuncs.push_back(bNewPar);
```

The polynomial can be constructed using regular arithmetic involving the parameters of the theory. Irrational numbers a/b can be represented as `ir(a,b)`. A trace function is available as `Tr()`. For the complex conjugate parameters, their β -functions must be included as:

```
BetaFunc bNewParDagger = Conjugate(bNewPar);
nameOfBetaFuncs.push_back(bNewParDagger);
```

RGIssearch needs separate **vectors** of β -functions for one-loop and two-loop, which must be in the same order. Finally, the algorithm can be initiated by calling:

```
findInvariants(1loopBetaFuncs, 2loopBetaFuncs);
```

3.4. Test Program

As an example, we consider a simple toy system of β -functions for three variables x, y, z :

$$\beta(x) = -4xy - 3y^2 + \frac{1}{16\pi^2}(-x + 6y), \quad (23a)$$

$$\beta(y) = -2x^2 + z + \frac{1}{16\pi^2}3x, \quad (23b)$$

$$\beta(z) = 6xy^2 + 4yz + \frac{1}{16\pi^2}(4xy - 3y^2 + z) \quad (23c)$$

This system has two invariants at two-loop level. To find them, the appropriate `equations.cpp` is shown below.

Using the default settings and enabling two-loop calculations, the output is:

```
-x^2 + 2y^2 - z + (1/16 pi^2){x + y}
-2y^3 - 2xz + (1/16 pi^2){x^2 + 3z}
```

These can easily be verified to be invariants of the above system.

Acknowledgements

Thanks to Ronald Kleiss and Wim Beekakker for suggestions and guidance along the way, and more so to Ronald Kleiss for carefully proofreading of the draft of this paper.

```
1 #include "src/common.h"
2 #include "src/invariants.h"
3
4 int main()
5 {
6     Param x("x", 1);
7     Param y("y", 1);
8     Param z("z", 1);
9
10    vector<BetaFunc> test_1;
11    vector<BetaFunc> test_2;
12
13    BetaFunc bx_1(x);
14    BetaFunc bx_2(x);
15    bx_1 = -4*x*y - 3*y*y;
16    bx_2 = -1*x + 6*y;
17    test_1.push_back(bx_1);
18    test_2.push_back(bx_2);
19
20    BetaFunc by_1(y);
21    BetaFunc by_2(y);
22    by_1 = -2*x*x + 1*z;
23    by_2 = 3*x;
24    test_1.push_back(by_1);
25    test_2.push_back(by_2);
26
27    BetaFunc bz_1(z);
28    BetaFunc bz_2(z);
29    bz_1 = 6*x*y*y + 4*y*z;
30    bz_2 = -4*x*y - 3*y*y + 1*z;
31    test_1.push_back(bz_1);
32    test_2.push_back(bz_2);
33
34    findInvariants(test_1, test_2);
35 }
```

References

- [1] J. Hetzel, W. Beenakker, Renormalisation group invariants and sum rules: fast diagnostic tools for probing high-scale physics, JHEP 1210 (2012) 176. [arXiv:1204.4336](#), [doi:10.1007/JHEP10\(2012\)176](#).
- [2] D. A. Demir, Renormalization group invariants in the MSSM and its extensions, JHEP 0511 (2005) 003. [arXiv:hep-ph/0408043](#), [doi:10.1088/1126-6708/2005/11/003](#).
- [3] M. Carena, P. Draper, N. R. Shah, C. E. Wagner, Determining the Structure of Supersymmetry-Breaking with Renormalization Group Invariants, Phys.Rev. D82 (2010) 075005. [arXiv:1006.4363](#), [doi:10.1103/PhysRevD.82.075005](#).
- [4] W. Beenakker, T. van Daal, R. Kleiss, R. Verheyen, Renormalization group invariants in supersymmetric theories: one- and two-loop results (to be published).[arXiv:1507.03470](#).
- [5] I. S. Duff, A. M. Erisman, J. K. Reid, Direct Methods for Sparse Matrices, Oxford University Press, Inc., New York, NY, USA, 1986.
- [6] W. S. Brown, The subresultant PRS algorithm, j-TOMS 4 (3) (1978) 237–249. [doi:http://dx.doi.org/10.1145/355791.355795](#).
- [7] <https://github.com/rbv/RGIsearch>.